Problem 1.a. Use the following encodings. Assume that the array index start from 1.
(0) **ROOT**() = index of root node = 1
(1) **CHILD**(i, j) = index of jth children of node $i = d(i-1)+j+1$, where $1 \leq j \leq d$
(2) **PARENT**(i) = index of parent of node $i = \lfloor \frac{i-2}{d} + 1 \rfloor$

Problem 1.b. $\Theta(log_d n)$

Problem 1.c. It takes $\Theta(dlog_d n)$. See the following procedures.

1: **procedure** MaxHeapify(A, i, n)
2:     $max\_idx := i$
3:     **for** $j = 1$ to d **do**
4:         $child\_idx =$ Child(i, j)
5:         **if** $child\_idx \leq n$ **and** $A[child\_idx] > A[max]$ **then**
6:             $max\_idx := j$
7:         **end if**
8:     **end for**
9:     **if** $max\_idx \neq i$ **then**
10:         $A[i], A[max\_idx] := A[max\_idx], A[i]$
11:         MaxHeapify(A, max_idx, n)
12:     **end if**
13: **end procedure**
14: **procedure** ExtractMax(A, n)
15:     $max := A[1]$; $A[1] := A[n]$
16:     MaxHeapify$(A, 1, n-1)$
17:     **return** A
18: **end procedure**

It takes $\Theta(d)$ times to execute from line 4 to line 8. Also, since the depth of the heap is $\Theta(log_d n)$, the line 11 could be executed at most $\Theta(log_d n)$ times.

Problem 1.d. It takes $\Theta(log_d n)$. See the following procedure and problem 1.e.

procedure Insert(A, key, n)
    $A[n+1] := -\infty$; IncreaseKey(A, n, key)
end procedure

Problem 1.e. It takes $\Theta(log_d n)$. See the following procedure.

procedure IncreaseKey(A, i, k)
    if $key < A[i]$ **then**
        error new key is smaller than current key
    end if
    $A[i] = key$
    while $i > 1$ **and** $A[$Parent$(i)] < A[i]$ **do**
        $A[i], A[$Parent$(i)] := A[$Parent$(i)], A[i]$; $i :=$ Parent(i)
    end while
end procedure

Since the depth of the heap is $\Theta(log_d n)$, the while loop could be executed at most $\Theta(log_d n)$ times.

Problem 2. For convinence, let **PART** as **PARTITION**, **QUICK** as **QUICKSORT**, **TRQ** as **TAIL-RECURSIVE-QUICKSORT**

Problem 2.a. r returns since line 4 is always executed. The following procedure is the modification.

```
1:  procedure PARTITION(A, p, r)
2:      x := A[r]
3:      i := p − 1; i′ = p − 1
4:      for k = p to r − 1 do
5:          if A[k] < x then
6:              i := i + 1; i′ = i′ + 1
7:              A[k], A[i] := A[i], A[k]
8:          else if A[k] = x then
9:              i′ = i′ + 1
10:             A[k], A[i′] := A[i′], A[k]
11:         end if
12:     end for
13:     A[i′ + 1], A[r] := A[r], A[i′ + 1]
14:     return ⌊(i + i′)/2⌋ + 1
15: end procedure
```

(1) Obviously, when $A[p \dots r]$ have the same value, only line 9 to 10 are executed, so the procedure returns $\lfloor (p-1+r-1)/2 \rfloor + 1 = \lfloor (p+r)/2 \rfloor$

(2) For correctness, claim that the following statement: $(\forall y \in A[p \dots i],\ y < A[r]) \wedge (\forall y \in A[(i+1) \dots i'],\ y = A[r]) \wedge (\forall y \in A[(i'+1) \dots k],\ y > A[r])$. Use induction on k. When $k = p$, the claim is obviously satisfied. Let the claim is satisfied when $k = \alpha - 1$ ($\alpha < r - 1$)

(i) If $A[\alpha] < x$, i and i' are increased by 1, $A[i]$ and $A[\alpha]$ are swapped, so $A[i] < x$, $A[\alpha] > x$ and others will be unchanged.

(ii) If $A[\alpha] = x$, i' is increased by 1, $A[i']$ and $A[\alpha]$ are swapped, so $A[i'] = x$, $A[\alpha] > x$ and others will be unchanged.

(iii) If $A[\alpha] > x$, nothing are changed.

(3) Also, since $A[(i+1) \dots i']$ are all equal to $A[r]$, select any element between them does not change its semantics in the return statement.

(4) $\therefore$ the modified **PART** is correct.

Problem 2.b. Since the **PART** method select the last element as a pivot, it always returns p, which makes **QUICK**(A, p, r) recursively calls only **QUICK**$(A, p+1, r)$. So, the recurrence relation becomes $T(n) = T(n-1) + \Theta(n)$ and $T(n) = \Theta(n^2)$.

Problem 2.c. QUICK does the followings: (i) check $p < r$, (ii) call **PART**(A, p, r), (iii) call **QUICK**$(A, p, q-1)$, (iv) call **QUICK**$(A, q+1, r)$. Since (iv) is the last operation of **QUICK**, it can be replaced by sequence from (i) to (iv), where p is replaced by $q + 1$. Therefore, without changing of its semantics, we can put step (i) to (iii) in a while loop, which its condition is same as (i), and update $p := q+1$ after each iteration. Then it becomes **TRQ**.

Problem 2.d. Let $A = [1, 2, \ldots, n]$
(1) **PART**(A, p, r) always returns r
(2) **TRQ**(A, p, r) always call **TRQ**$(A, p, r-1)$
(3) $\therefore$ the stack will be filled from **TRQ**$(A, 1, n)$ to **TRQ**$(A, 1, 1)$

Problem 2.e. See the following procedure.

```
procedure TRQ(A, p, r)
    while p < r do
        q = Partition(A, p, r)
        if q − p ≥ r − q then
            TRQ(A, q + 1, r); r := q − 1
        else
            TRQ(A, p, q − 1); p := q + 1
        end if
    end while
end procedure
```

In problem 2.d, the depth of the stack was $\Theta(n)$ because the value of $r - p$ reduces only one. To achieve $\Theta(\lg n)$ stack depth, it needs to call **TRQ** with new r' and p' values which $r' - p' \leq (r-p)/2$. Therefore, modified **TRQ** should not choose the left partition, but choose the small partition between left and right.
The running time is not effected since the semantics is actually same.